Table of Contents

Porting & Developing Applications

Porting & Developing: Overview

DRAFT

This article is being reviewed for completeness and technical accuracy.

When you are in the process of developing a code or porting a code from another platform, it is important that the code runs correctly and/or reproduces the results from another platform.

These are some steps you can follow when developing or porting a code or when testing a new version of a compiler.

General guidelines:

- Start with small problem sizes and a few time steps/iterations so that you won't have to wait in the queue for a long time just to check whether the program is running correctly. Setting up your PBS script, data files, and getting the program to run correctly can often be done with 10 minute jobs.
- Use PBS' debug queue to get better turn-around time (q=debug).
- While porting, make the fewest changes possible in the code.
- Use the same data sets to compare results on both old and new platforms.
- Don't assume that an absence of error messages means the program is running correctly on either the old or the new platforms.
- Be attentive to porting user data files. Fortran FORM='unformatted' files cannot be assumed to be portable.
- Don't assume that the new platform is wrong and the old platform is right. Both might be wrong.

Other useful information that helps you to port or develop a code on NAS HECC systems can be found in subsequent articles.

Endian and Related Environment Variables or Compiler Options

DRAFT

This article is being reviewed for completeness and technical accuracy.

Intel Fortran expects numeric data, both integer and floating-point data, to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

If your program needs to read or write unformatted data files that are not in little endian order, you can use one of the six methods (listed in the order of precedence) provided by Intel below.

1. Set an environment variable for a specific unit number before the file is opened. The environment variable is named **FORT_CONVERTn**, where n is the unit number. For example:

   ```
   setenv FORT_CONVERT28 BIG_ENDIAN
   ```

   No source code modification or recompilation is needed.

2. Set an environment variable for a specific file name extension before the file is opened. The environment variable is named **FORT_CONVERT.ext** or **FORT_CONVERT_ext**, where ext is the file name extension (suffix). The following example specifies that a file with an extension of .dat is in big endian format:

   ```
   setenv FORT_CONVERT.DAT BIG_ENDIAN
   ```

   Some Linux command shells may not accept a dot (.) for environment variable names. In that case, use FORT_CONVERT_ext instead.

   No source code modification or recompilation is needed.

3. Set an environment variable for a set of units before any files are opened. The environment variable is named **F_UFMTENDIAN**.

   Syntax:

   Csh: setenv F_UFMTENDIAN MODE;EXCEPTION

   Sh : export F_UFMTENDIAN=MODE;EXCEPTION

MODE = big | little

EXCEPTION = big:ULIST | little:ULIST | ULIST

ULIST = U | ULIST,U

U = decimal | decimal-decimal

MODE defines the current format of the data, represented in the files; it can be omitted. The keyword "little" means that the data have little- endian format and will not be converted. For IA-32 systems, this keyword is a default. The keyword "big" means that the data have big endian format and will be converted. This keyword may be omitted together with the colon.

EXCEPTION is intended to define the list of exclusions for MODE; it can be omitted. EXCEPTION keyword (little or big) defines data format in the files that are connected to the units from the EXCEPTION list. This value overrides MODE value for the units listed.

Each list member U is a simple unit number or a number of units. The number of list members is limited to 64. decimal is a non-negative decimal number less than 2**32.

The environment variable value should be enclosed in quotes if the semicolon is present.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Example:

- setenv F_UFMTENDIAN big

  All input/output operations perform conversion from big-endian to little-endian on READ and from little-endian to big-endian on WRITE.

- setenv F_UFMTENDIAN "little;big:10,20"

  or setenv F_UFMTENDIAN big:10,20

  or setenv F_UFMTENDIAN 10,20

  In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

- setenv F_UFMTENDIAN "big;little:8"

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

- setenv F_UFMTENDIAN 10-20

  Define 10, 11, 12, ...19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

4. Specify the **CONVERT** keyword in the OPEN statement for a specific unit number. Note that a hard-coded OPEN statement CONVERT keyword value cannot be changed after compile time. The following OPEN statement specifies that the file graph3.dat is in VAXD unformatted format:

   ```
   OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED',
   UNIT=15)
   ```

5. Compile the program with an **OPTIONS** statement that specifies the CONVERT=keyword qualifier. This method affects all unit numbers using unformatted data specified by the program. For example, to use VAX F_floating and G_floating as the unformatted file format, specify the following OPTIONS statement:

   ```
   OPTIONS /CONVERT=VAXG
   ```

6. Compile the program with the command-line **-convert keyword** option, which affects all unit numbers that use unformatted data specified by the program. For example, the following command compiles program file.for to use VAXD floating-point data for all unit numbers:

   ```
   ifort file.for -o vconvert.exe -convert vaxd
   ```

In addition, if the record length of your unformatted data is in byte units (Intel Fortran default is in word units), use the **-assume byterecl** compiler option when compiling your source code.

OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for various platforms.

Intel version 11.x compilers support OpenMP spec-3.0 while 10.x compilers support spec-2.5.

Building OpenMP Applications

The following Intel compiler options can be used for building or analyzing OpenMP applications:

- *-openmp*

  Enables the parallelizer to generate multithreaded code based on OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. The *-openmp* option works with both *-O0* (no optimization) and any optimization level of *-O*. Specifying *-O0* with *-openmp* helps to debug OpenMP applications.

  Note that setting *-openmp* also sets *-automatic*, which causes all local, non-SAVEd variables to be allocated to the run-time stack, which may provide a performance gain for your applications. However, if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. If you want to cause variables to be placed in static memory, specify option *-save*. If you want only scalar variables of certain intrinsic types (integer, real, complex, logical) to be placed on the run-time stack, specify option *-auto-scalar*.
- *-assume cc_omp* or *-assume nocc_omp*

  -assume cc_omp enables conditional compilation as defined by the OpenMP Fortran API. That is, when "!$space" appears in free-form source or "c$spaces" appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.

  -assume nocc_omp tells the compiler that conditional compilation as defined by the OpenMP Fortran API is disabled unless option -openmp (Linux) or /Qopenmp (Windows) is specified.

- *-openmp-lib legacy* or *-openmp-lib compat*

  Choosing *-openmp-lib legacy* tells the compiler to use the legacy OpenMP run-time

library (*libguide*). This setting does not provide compatibility with object files created using other compilers. This is the default for Intel version 10.x compilers.

Choosing *-openmp-lib compat* tells the compiler to use the compatibility OpenMP run-time library (*libiomp*). This is the default for Intel version 11.x compilers.

On Linux systems, the compatibility Intel OpenMP run-time library lets you combine OpenMP object files compiled with the GNUgcc or gfortran compilers with similar OpenMP object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

You cannot link object files generated by the Intel® Fortran compiler to object files compiled by the GNU Fortran compiler, regardless of the presence or absence of the *-openmp* (Linux) or /Qopenmp (Windows) compiler option. This is because the Fortran run-time libraries are incompatible.

NOTE: The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compiler earlier than 10.0.

- *-openmp-link dynamic* or *-openmp-link static*

  Choosing *-openmp-link dynamic* tells the compiler to link to dynamic OpenMP run-time libraries. This is the default for Intel version 11.x compilers.

  Choosing *-openmp-link static* tells the compiler to link to static OpenMP run-time libraries.

  Note that the compiler options *-static-intel* and *-shared-intel* have no effect on which OpenMP run-time library is linked.

  Note that this option is only available for newer Intel compilers (version 11.x).
- *-openmp-profile*

  Enables analysis of OpenMP applications. To use this option, you must have Intel(R) Thread Profiler installed, which is one of the Intel(R) Threading Tools. If this threading tool is not installed, this option has no effect.

  Note that Intel Thread Profiler is not installed on Pleiades.

- *-openmp-report[n]*

  Controls the level of diagnostic messages of the OpenMP parallelizer. n=0,1,or 2.

- *-openmp-stub*

  Enables compilation of OpenMP programs in sequential mode. The OpenMP

directives are ignored and a stub OpenMP library is linked.

OpenMP Environment Variables

There are a few OpenMP environment variables one can set. The most commonly used are:

- *OMP_NUM_THREADS num*

  Sets number of threads for parallel regions. Default is 1 on Pleiades. Note that you can use *ompthreads* in the PBS resource request to set values for OMP_NUM_THREADS. For example:

```
%qsub -I -lselect=1:ncpus=4:ompthreads=4
Job 991014.pbspl1.nas.nasa.gov started on Sun Sep 12 11:33:06 PDT 2010
...
PBS r3i2n9> echo $OMP_NUM_THREADS
4
PBS r3i2n9>
```

- *OMP_SCHEDULE type[,chunk]*

  Sets the run-time schedule type and chunk size. Valid OpenMP schedule types are *static, dynamic, guided,* or *auto*. Chunk is a positive integer.

- *OMP_DYNAMIC true* or *OMP_DYNAMIC false*

  Enables or disables dynamic adjustment of threads to use for parallel regions.

- *OMP_STACKSIZE size*

  Specifies size of stack for threads created by the OpenMP implementation. Valid values for size (a positive integer) are *size, size*B*, size*K*, size*M*, size*G. If units B, K, M or G are not specified, size is measured in kilobytes (K).

  Note that this feature is included in OpenMP spec-3.0, but not in spec-2.5.

Note that Intel also provides a few additional environment variables. The most commonly used are:

- *KMP_AFFINITY type*

  Binds OpenMP threads to physical processors. Avaiable *type*: *compact, disabled, explicit, none, scatter*. For more information on the various types, see this Intel web page.

  There is a conflict between *KMP_AFFINITY* in Intel 11.x runtime

library and *dplace*, causing all threads to be placed on a single CPU when both are used. It is recommended that *KMP_AFFINITY* be set to *disabled* when using *dplace*.

- *KMP_MONITOR_STACKSIZE*

  Sets stacksize in bytes for monitor thread.

- *KMP_STACKSIZE*

  Sets stacksize in bytes for each thread.

For more information, please see the official OpenMP web site.

Compilers

Intel Compiler

DRAFT

This article is being reviewed for completeness and technical accuracy.

Intel compilers are recommended for building your applications on either Pleiades or Columbia.

On Columbia, a system default version has been loaded automatically. On Pleiades, there is no system default--you must load a specific module. Use the "module avail" command on Pleiades to see what versions are available and load an Intel compiler module before compiling. For example:

```
% module load comp-intel/11.1.072
```

Notice that when a compiler module is loaded, some environment variables, such as FPATH, INCLUDE, LD_LIBRARY_PATH, etc., are set or modified to add the paths to certain commands, include files, or libraries, to your environment. This helps to simplify the way you do your work.

To check what environmant variables will be modified for a module, do, for example:

```
% module show comp-intel/11.1.072
```

On Columbia and Pleiades, there are Intel compilers for both Fortran and C/C++:

- **Intel Fortran Compiler: ifort (version 8 and above)**

  The ifort command invokes the Intel(R) Fortran Compiler to preprocess, compile, assemble, and link Fortran programs.

  ```
  % ifort [options] file1 [file2 ...]
  ```

  Read **man ifort** for all available compiler options.

  To see the compiler options by categories, do:

  ```
  % ifort -help
  ```

file*N* is a Fortran source (.f .for .ftn .f90 .fpp .F .FOR .F90 .i .i90), assembly (.s .S), object (.o), static library (.a), or other linkable file.

Source Files Suffix Interpretation:

- .f, .for, or .ftn : fixed-form source files
- .f90 : free-form F95/F90 source files
- .fpp, .F, .FOR, .FTN, or .FPP: fixed-form source files which must be preprocessed by the fpp preprocessor before being compiled
- .F90 : free-form Fortran source files which must be pre-pro- cessed by the fpp preprocessor before being compiled

- **Intel C/C++ compiler: icc and icpc (version 8 and above)**

The Intel(R) C++ Compiler is designed to process C and C++ programs on Intel-architecture-based systems. You can preprocess, compile, assemble, and link these programs.

```
% icc [options] file1 [file2 ...]
% icpc [options] file1 [file2 ...]
```

Read **man icc** for all available compiler options.

To see the compiler options by categories, do:

```
% icc -help
```

```
The icpc command uses the same compiler options as the icc
command. Invoking the compiler using icpc compiles .c, and .i
files as C++. Invoking the compiler using icc compiles .c and
.i files as C. Using icpc always links in C++ libraries. Using
icc only links in C++ libraries if C++ source is provided on
the command line.
```

file*N* represents a C/C++ source (.C .c .cc .cp .cpp .cxx .c++ .i), assembly (.s), object (.o), static library (.a), or other linkable file.

GNU Compiler Collection

DRAFT

This article is being reviewed for completeness and technical accuracy.

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for several major programming languages. These languages currently include C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada.

The GNU C and C++ compiler (gcc and g++) and Fortran compiler (gfortran) through the Linux OS distribution are available on Pleiades and Columbia. The current version installed (under /usr/bin) can be found with the following command:

```
% gcc -v
 ... gcc version 4.1.2 20070115 (SUSE Linux)
```

Newer versions of GNU compilers can be requested and installed as modules. Currently, there is a gcc/4.4.4 module, which includes gcc, g++, and gfortran, available on Pleiades.

Read **man gcc** and **man gfortran** for more information.

MPI Libraries

SGI MPT

DRAFT

This article is being reviewed for completeness and technical accuracy.

SGI's Message Passing Interface (MPI) is a component of the Message Passing Toolkit (MPT), which is a software package that supports parallel programming across a network of computer systems through a technique known as message passing. It requires the presence of an Array Services daemon (arrayd) on each host to run MPI processes.

SGI's MPT 1.x versions support the MPI 1.2 standard and certain features of MPI-2. The 2.x versions will be fully MPI-2 compliant.

On Columbia, the current system default version is mpt.1.16. A 2.x version will be available when the operating system is upgraded to SGI ProPack 7SP1.

On Pleiades, there is no default version. You can enable the recommended version, mpt.2.04.10789, by:

```
%module load mpi-sgi/mpt.2.04.10789
```

Note that certain environment variables are set or modified when an MPT module is loaded. To see what variables are set when a module is loaded (for example, mpi-sgi/mpt.2.04.10789), do:

```
%module show mpi-sgi/mpt.2.04.10789
```

To build an MPI application using SGI's MPT, use a command such as one of the following:

```
%ifort -o executable_name prog.f -lmpi
%icc -o executable_name prog.c -lmpi
%icpc -o executable_name prog.cxx -lmpi++ -lmpi
%gfortran -I/nasa/sgi/mpt/1.26/include -o executable_name prog.f -lmpi
%gcc  -o executable_name prog.c -lmpi
%g++ -o executable_name prog.cxx -lmpi++ -lmpi
```

MVAPICH

DRAFT

This article is being reviewed for completeness and technical accuracy.

MVAPICH is open source software developed largely by the Network-Based Computing Laboratory (NBCL) at Ohio State University. MVAPICH develops the Message Passing Interface (MPI) style of process-to-process communications for computing systems employing InfiniBand and other Remote Direct Memory Access (RDMA) interconnects.

MVAPICH software is typically used across the network of a cluster computer system for improved performance and scalability of applications.

MVAPICH is an MPI-1 implementation while MVAPICH2 is an MPI-2 implementation (conforming to MPI 2.2 standard) which includes all MPI-1 features.

MVAPICH1/MVAPICH2 are installed on Pleiades, but not Columbia. You must load in an MVAPICH1 or MVAPICH2 module before using it. For example:

```
%module load mpi-mvapich2/1.4.1/intel
```

A variety of MPI compilers, such as mpicc, mpicxx, mpiCC, mpif77, or mpif90, are provided in each MVAPICH/MVAPICH2 distribution. The correct compiler should be selected depending on the programming language of your MPI application.

To build an MPI application using MVAPICH1/MVAPICH2:

```
%mpif90 -o executable_name prog.f
%mpicc -o executable_name prog.c
```

Math & Scientific Libraries

MKL

DRAFT

This article is being reviewed for completeness and technical accuracy.

The Intel Math Kernel Library (MKL) is composed of highly optimized mathematical functions for engineering and scientific applications requiring high performance on Intel platforms. The functional areas of the library include linear algebra consisting of LAPACK and BLAS, fast Fourier transform (FFT), and vector transcendental functions.

MKL release 10.x is part of the Intel compiler 11.0 and 11.1 releases. Once you load in a 11.x compiler module, the path to the MKL library is automatically included in your default path. If you choose to use Intel compiler 10.x or earlier versions, you have to load an MKL module separately.

A Layered Model for MKL

Starting with MKL release 10.0, Intel employs a layered model for the MKL library. The layers are:

- Interface layer
  - LP64 interface (uses 32-bit integer type) or ILP64 interface (uses 64-bit integer type)
  - SP2DP interface

    which supports Cray-style naming in applications targeted for the Intel 64 or IA-64 architecture and using the ILP64 interface. SP2DP interface provides a mapping between single-precision names (for both real and complex types) in the application and double-precision names in Intel MKL BLAS and LAPACK.
- Threading layer
  - sequential

    The sequential (non-threaded) mode requires no Compatibility OpenMP* or Legacy OpenMP* run-time library, and does not respond to the environment variable OMP_NUM_THREADS or its Intel MKL equivalents. In this mode, Intel MKL runs unthreaded code. However, it is thread-safe, which means that

you can use it in a parallel region from your own OpenMP code. You should use the library in the sequential mode only if you have a particular reason not to use Intel MKL threading. The sequential mode may be helpful when using Intel MKL with programs threaded with some non-Intel compilers or in other situations where you may, for various reasons, need a non-threaded version of the library (for instance, in some MPI cases).

Note that the *sequential.* library depends on the POSIX threads library (pthread), which is used to make the Intel MKL software thread-safe and should be listed on the link line.

- threaded

The *threaded* library in MKL version 10.x supports the implementation of OpenMP that many compilers (Intel, PGI, GNU) provide.

- Computational layer

For any given processor architecture (IA-32, IA-64, or Intel(R) 64) and OS, this layer has only one computational library to link with, regardless of the Interface and Threading layer.

- Compiler Support Run-time libraries

  - libiomp

    Intel(R) Compatibility OpenMP run-time library

  - libguide

    Intel(R) Legacy OpenMP run-time library

For example, to do a dynamic linking of myprog.f and parallel Intel MKL supporting LP64 interface, use:

```
ifort myprog.f -Wl,--start-group -lmkl_intel_lp64 \
-lmkl_intel_thread -lmkl_core -Wl,--end-group -openmp
```

If you are unsure of what MKL libraries to link with, use the suggestion provided in this Intel web site by providing the proper OS (e.g. Linux), processor architecture (e.g. Intel(R) 64), compiler (e.g. Intel or Intel Compatible), dynamic or static linking, integer length, sequential or multi-threaded, OpenMP library, cluster library (e.g. BLACS, ScaLAPACK), MPI library (Intel MPI, MPICH2, SGIMPT, etc.).

The -mkl Switch of Intel Compiler Version 11.1

Starting from Intel compiler version 11.1, a -mkl switch is provided to link to certain parts of the MKL library.

```
-mkl[=]
          link to the Intel(R) Math Kernel Library (Intel(R) MKL) and
          bring in the associated headers
            parallel   - link using the threaded Intel(R) MKL libraries.
                         This is the default when -mkl is specified
            sequential - link using the non-threaded Intel(R) MKL libraries
            cluster    - link using the Intel(R) MKL Cluster libraries plus
                         the sequential Intel(R) MKL libraries
```

The libraries that are linked in for

* -mkl=parallel

```
      --start-group \
      -lmkl_solver_lp64 \
      -lmkl_intel_lp64 \
      -lmkl_intel_thread \
      -lmkl_core \
      -liomp5 \
      --end-group \
```

* -mkl=sequential

```
      --start-group \
      -lmkl_solver_lp64_sequential \
      -lmkl_intel_lp64 \
      -lmkl_sequential \
      -lmkl_core \
      --end-group \
```

* -mkl=cluster

```
      --start-group \
      -lmkl_solver_lp64 \
      -lmkl_intel_lp64 \
      -lmkl_cdft_core \
      -lmkl_scalapack_lp64 \
      -lmkl_blacs_lp64 \
      -lmkl_sequential \
      -lmkl_core \
      -liomp5 \
      --end-group \
```

Where to find more information about MKL

Man pages and two PDF files from Intel are available for each version of MKL.

- Man pages of Intel MKL

  A collection of man pages of Intel MKL functions are available under the man3 subdirectory (e.g., /nasa/intel/Compiler/11.1/072/man/en_US/man3) of the MKL installation. You will have to load an MKL module or an Intel compiler 11.x module before you can see the man pages. For example,

```
% module load comp-intel/11.1.072
% man gemm
```

provides information about [s,d,c,z,sc,dz]gemm routines.

Unfortunately, there does not appear to be a 'man mkl' page.

- Intel MKL Reference Manual (mklman.pdf)

  Contains detailed descriptions of the functions and interfaces for all library domains:

  - BLAS
  - LAPACK
  - ScaLAPACK
  - Sparse Solver
  - Interval Linear Solvers
  - Vector Math Library (VML)
  - Vector Statistical Library (VSL)
  - Conventional DFTs and Cluster DFTs
  - Partial Differential Equations support
  - Optimization Solvers

- Intel MKL User's Guide (userguide.pdf)

  Provides Intel MKL usage information in greater detail:

  - getting started information
  - application compiling and linking depending on a particular platform and function domain
  - building custom DLLs
  - configuring the development environment
  - coding mixed-language calls
  - threading
  - memory management
  - ways to obtain best performance

  The two pdf files can be found in the 'doc' or 'Documentation' directory of the MKL installation. For example, on Pleiades,

  - MKL version 10.0.011

    /nasa/intel/mkl/10.0.011/doc

  - The version included in the Intel compiler module 11.1.072

    /nasa/intel/Compiler/11.1/072/Documentation/en_US/mkl

SCSL

DRAFT

This article is being reviewed for completeness and technical accuracy.

SCSL is a comprehensive collection of scientific and mathematical functions that have been optimized for use on the Altix systems such as Columbia . The libraries include optimization of basic linear algebra subprograms (BLAS), a linear algebra package, signal processing functions such as fast Fourier transforms (FFTs), and liner filtering operations and other basic solver functions. More information can be found through 'man scsl'.

Starting with ProPack 5, SCSL is no longer supported by SGI. Although SCSL is still available on Columbia (but not on Pleiades), users are recommended to use Intel MKL instead.
SCSL version(s) available on Columbia systems:

- scsl.1.5.0.0 (does not work properly with intel-comp.9.1.039)
- scsl.1.5.1.0
- scsl.1.5.1.1 (contains Scalapack in libsdsm.so)
- scsl.1.6.1.0

To use SCSL, link one of the following libraries:

```
-lscs
-lscs_mp      (for multi-threaded programs)
-lscs_i8
-lscs_i8_mp
```

MKL FFTW Interface

DRAFT

This article is being reviewed for completeness and technical accuracy.

Some users have installed the FFTW library in their own directory (for example, /u/user/bin/fftw) and would link to the FFTW library as follows:

```
ifort -O3
  -I/u/user/bin/fftw/include   \
  -o fftw_xmpl.exe fftw_xmpl.f \
  -L/u/user/bin/fftw/lib -lfftw3
```

An MKL FFTW interface has been created for Intel compiler version 11.0.083 and later versions. Users no longer have to keep their own copy of FFTW. Follow these steps to use the MKL FFTW interface:

- Load a compiler module 11.0.083 or a later version such as comp-intel/11.1.072

```
module load comp-intel/11.1.072
```

- Compile and link

```
ifort -O3                                              \
  -I/nasa/intel/Compiler/11.1/072/mkl/include/fftw     \
  -o fftw_xmpl.exe fftw_xmpl.f                         \
  -lfftw3xf_intel -lmkl_intel_lp64 -lmkl_intel_thread \
  -lmkl_core -lguide
```

Program Development Tools

Recommended Intel Compiler Debugging Options

DRAFT

This article is being reviewed for completeness and technical accuracy.

- Commonly used options for debugging:

  -O0
  : Disables optimizations. Default is -O2

  -g
  : Produces symbolic debug information in object file (implies -O0 when another optimization option is not explicitly set)

  -traceback
  : Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run-time.

    ```
    Specifying -traceback will increase the size of the
    executable program, but has no impact on run-time
    execution speeds.
    ```

  -check all
  : Checks for all run-time failures. **Fortran only.**

  -check bounds
  : Alternate syntax: -CB. Generates code to perform run-time checks on array subscript and character substring expressions. **Fortran only.**

    ```
    Once the program is debugged, omit this option to reduce
    executable program size and slightly improve run-time
    performance.
    ```

  -check uninit
  : Checks for uninitialized **scalar** varaibles without the SAVE attribute. **Fortran only.**

  -check-uninit
  : Enables run-time checking for uninitialized variables. If a variable is read before it is written, a run-time error routine will be called. Run-time checking of

undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays. **C/C++ only**.

-ftrapuv
: Traps uninitialized variables by setting any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors. This option sets -g.

-debug all
: Enables debug information and control output of enhanced debug information. To use this option, you must also specify the -g option.

-gen-interfaces -warn interfaces
: Tells the compiler to generate an interface block for each routine in a source file; the interface block is then checked with -warn interfaces

- Options for handling floating-point exceptions:

-fpe{0|1|3}
: Allows some control over floating-point exception (divide by zero, overflow, invalid operation, underflow, denormalized number, positive infinity, negative infinity or a NaN) handling for the **main program** at run-time. **Fortran only.**

  - -fpe0: underflow gives 0.0; abort on other IEEE exceptions
  - -fpe3: produce NaN, signed infinities, and denormal results

  Default is -fpe3 with which all floating-point exceptions are disabled and floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. Use of -fpe3 on IA-64 systems such as Columbia will slow run-time performance.

-fpe-all={0|1|3}
: Allows some control over floating-point exception handling for **each routine** in a program at run-time. Also sets -assume ieee_fpe_flags. Default is -fpe-all=3. **Fortran only.**

-assume ieee_fpe_flags
: Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance. **Fortran only.**

-ftz

Flushes denormal results to zero when the application is in the gradual underflow mode. This option has effect only when compiling the **main program**. It may improve performance if the denormal values are not critical to your application's behavior. For IA-64 systems (such as Columbia), -O3 sets -ftz. For Intel 64 systems (such as Pleiades), every optimization option O level, except -O0, sets -ftz.

- Options for handling floating-point precision:

-mp
: Enables improved floating-point consistency during calculations. This option limits floating-point optimizations and maintains declared precision. -mp1 restricts floating-point precision to be closer to declared precision. It has some impact on speed, but less than the impact of -mp.

-fp-model precise
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance.

-fp-model strict
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

-fp-speculation=off
: Disables speculation of floating-point operations. Default is -fp-speculation=fast

-pc{64|80}
: For Intel EM64 only. Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the -pc option. -pc64 sets internal FPU precision to 53-bit significand. -pc80 is the default and it sets internal FPU precision to 64-bit significand.

Totalview

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is a GUI-based debugging tool that gives you control over processes and thread execution and visibility into program state and variables for C, C++ and Fortran applications. It also provides memory debugging to detect errors such as memory leaks, deadlocks and race conditions, etc.

Totalview allows you to debug serial, OpenMP, or MPI codes.

Totalview is available on both Pleiades and Columbia. See Totalview Debugging on Pleiades for some basic instructions on how to start using Totalview on Pleiades.

See Totalview Debugging on Columbia for some basic instructions on how to start using Totalview on Columbia.

Totalview Debugging on Pleiades

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is an advanced debugger for complex and parallel codes. Its versions have been installed as modules. To find out what versions of totalview are available, use the 'module avail' command.

There are additional steps needed in order to start the TotalView GUI. You'll need to rely on the ForwardX11 feature of your ssh. First, you'll have to make sure that your sysadmin had turned on ForwardX11 when SSH was installed on your system or use the -X or -Y (if available) options of ssh to enable X11 forwarding for your SSH session.

For debugging on a back-end node, do:

- Compile your code with -g
- Start a PBS session. For example:

```
% qsub -I -V -lselect=2:ncpus=8,walltime=1:00:00
```

- Test the X11 forwarding with xlock

```
% xclock
```

- Load the totalview module

```
% module load apps/etnus/totalview.8.6.2-1
```

- Set the environment variable TOTALVIEW

```
% setenv TOTALVIEW `which totalview` (for csh users)
```
or
```
% export TOTALVIEW=`which totalview` (for bash users)
```

- Start TotalView debugging
    - ♦ For serial applications:
        - ◊ Simply start totalview with your application as an argument

```
% totalview ./a.out
```

If your application requires arguments:

```
% totalview ./a.out -a app_arg_1 app_arg_2
```

- For MPI applications:

1. Make sure you load the appropriate modules, including the compiler, and mpi module. For example:

   For applications built with SGI's MPT, make sure that you have loaded the latest MPT module:

   ```
   % module load comp-intel/11.1.072
   % module load mpi-sgi/mpt.1.26
   ```

   For applications built with MVAPICH:

   ```
   % module load comp-intel/11.1.072
   % module load mpi-mvapich2/1.4.1/intel
   ```

2. Launch totalview by typing "totalview" all by itself. Once the totalview windows pop up, you will see four tabs in the "New Program" window: Program, Arguments, Standard I/O and Parallel.

3. Fill in the executable name in the "Program" box or use the Browse button to find the executable

4. Give any arguments to your executable by clicking on the "Arguments" tab and filling in what you need. If you need to redirect input from a file, do so by clicking the "Standard I/O" tab and filling in what you need.

5. In the "Parallel" tab, select the parallel system option MVAPICH2 or mpt_1.26 depending on which version of MPI you have compiled with.

6. Enter in the number of processes in the 'tasks' box; leave the 'nodes' field 0. For example, if you run your application with 2 nodes x 4 MPI processes/node = 8 processes in total, fill in 8 in the 'tasks' box and 0 in the 'node' box.

7. Then press "Go" to start. Note that it may initially dump you into the mpiexec assembler source which is not your own code.

8. Respond to the popup dialog box which says "Process xxx is a parallel job. Do you want to stop the job now?" Choose "No" if you just want to run your application. Choose "Yes" if you want to set breakpoint in your source code or do other tasks before running.

More information about TotalView can be found at the Totalview online documentation website.

Totalview Debugging on Columbia

DRAFT

This article is being reviewed for completeness and technical accuracy.

TotalView is an advanced debugger for complex and parallel codes. It has been installed as modules. To find out what versions of totalview are available, use the command 'module avail totalview'.

You'll need to rely on the ForwardX11 feature of your ssh. First, you'll have to make sure that your sysadmin had turned on ForwardX11 when SSH was installed on your local system or use the -X or -Y (if available) options of ssh to enable X11 forwarding for your SSH session.

For debugging on the front-end cfe2, do:

- Login to the front-end cfe2
- Compile your code with -g
- Make sure that X11 forwarding works and test it with xclock

```
cfe2%echo $DISPLAY
cfe2:xx.0
cfe2%xclock
```

- Load the totalview module

```
cfe2% module load totalview.8.9.0-1
```

- Start totalview. For serial jobs:

  cfe2% totalview a.out

  For MPI jobs built with SGI's MPT library:

  cfe2% totalview mpirun.real -a -np xxx a.out

For debugging on a back-end node, do:

- Compile your code with -g
- Start a PBS session and pass in the environment variable DISPLAY. Assuming PBS assign your job to run on Columbia21

```
cfe2% qsub -I -v DISPLAY -lncpus=8,walltime=1:00:00
```

- Test the X11 forwarding with xlock

```
PBS(8cpus)columbia21% xclock
```

- Load the totalview module

```
PBS(8cpus)columbia21% module load totalview.8.9.0-1
```

- Start totalview. For serial jobs:

  PBS(8cpus)columbia21% totalview a.out

  For MPI jobs built with SGI's MPT library:

  PBS(8cpus)columbia21% totalview mpirun.real -a -np xxx a.out

More information about TotalView can be found at the Totalview online documentation website.

IDB

DRAFT

This article is being reviewed for completeness and technical accuracy.

The Intel Debugger is a symbolic source code debugger that debugs programs compiled by the Intel Fortran and C/C++ Compiler, and the GNU compilers (gcc, g++).

IDB is included in the Intel compiler distribution. For IA-64 systems such as Columbia, both the Intel 10.x and 11.x compiler distributions provide only an IDB command-line interface. To use IDB on Columbia, load an Intel 10.x or 11.x compiler module. For example:

```
%module load intel-comp.11.1.072
%idb
(idb)
```

For Intel 64 systems such as Pleiades, a command-line interface is provided in the 10.x distribution and is invoked with the command *idb* just like on Columbia. For the Intel 11.x compilers, both a graphical user interface (GUI), which requires a Java Runtime, and a command-line interface are provided. The command *idb* invokes the GUI interface by default. To use the command-line interface under 11.x compilers, use the command *idbc*. For example:

```
%module load comp-intel/11.1.072 jvm/jre1.6.0_20
%idb
.... This will bring up an IDB GUI ....

%module load comp-intel/11.1.072
%idbc
(idb)
```

Be sure to compile your code with the *-g* option for symbolic debugging.

Depending on the Intel compiler distributions, the Intel Debugger can operate in either the gdb mode, dbx mode or idb mode. The available commands under these modes are different.

For information on IDB in the 10.x distribution, read **man idb**.

For information on IDB in the 11.x distribution, read documentations under *pfe or cfe2:/nasa/intel/Compiler/11.1/072/Documentation/en_US/idb*

GDB

DRAFT

This article is being reviewed for completeness and technical accuracy.

The GNU Debugger, GDB, is available on both Pleiades and Columbia under /usr/bin. It can be used to debug programs written in C, C++, Fortran and Modula-a.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Be sure to compile your code with *-g* for symbolic debugging.

GDB is typically used in the following ways:

- Start the debugger by itself
```
%gdb
(gdb)
```

- Start the debugger and specify the executable
```
%gdb your_executable
(gdb)
```

- Start the debugger, and specify the executable and core file
```
%gdb your_executable core-file
(gdb)
```

- Attach gdb to a running process
```
%gdb your_executable pid
(gdb)
```

At the prompt (gdb), type in commands such as *break* for setting a breakpoint, *run* for starting to run your executable, *step* for stepping into next line, etc. Read **man gdb** to learn more on using gdb.

Using pdsh_gdb for Debugging Pleiades PBS Jobs

DRAFT

This article is being reviewed for completeness and technical accuracy.

A script called *pdsh_gdb*, created by NAS staff Steve Heistand, is available on Pleiades under */u/scicon/tools/bin* for debugging PBS jobs **while the job is running**.

Launching this script from a Pleiades front-end node allows one to connect to each compute node of a PBS job and create a stack trace of each process. The aggregated stack trace from each process will be written to a user specified directory (by default, it is written to ~/tmp).

Here is an example of how to use this script:

```
pfe1% mkdir tmp
pfe1% /u/scicon/tools/bin/pdsh_gdb -j jobid -d tmp -s -u nas_username
```

More usage information can be found by launching pdsh_gdb without any option:

```
pfe1% /u/scicon/tools/bin/pdsh_gdb
```

Porting to Pleiades

Recommended compiler options

DRAFT

This article is being reviewed for completeness and technical accuracy.

Intel compiler versions 10.0, 10.1, 11.0, 11.1, and 12.0 are available on Pleiades as modules. Use the 'module avail' command to find available versions. Since NAS does not set a default version for users on Pleiades, be sure to use the 'module load ...' command to load the version you want to use.

In addition to the few flags mentioned in the article Recommended Intel Compiler Debugging Options, here are a few more to keep in mind:

Turn on optimization: *-O3*

If you do not specify an optimization level (*-On*, *n=0,1,2,3*), the default is *-O2*. If you want more aggressive optimizations, you can use *-O3*. Note that using *-O3* may not improve performance for some programs.

Generate optimized code for a processor type: *-xS, -xSSE4.1* or *-xSSE4.2*

Intel version 10.x, 11.x and 12.x compilers provide flags for generating optimized codes specialized for various instruction sets used in specific processors or microarchitectures.

Processor Type	Intel V10.x	Intel V11.x and above
Harpertown	-xS	-xSSE4.1
Nehalem-EP		
	N/A	-xSSE4.2
Westmere-EP		

Since the instruction set is upward compatible, an application which is compiled with *-xSSE4.1* can run on either Harpertown or Nehalem-EP or Westmere-EP processors. An application which is compiled with *-xSSE4.2* can run ONLY on Nehalem-EP and Westmere-EP processors.

If your goal is to get the best performance out of the Nehalem-EP/Westmere-EP processors, it is recommended that you do the following:

- Use either Intel 11.x or 12.x compilers as they are designed for

Nehalem-EP/Westmere-EP micro-architecture optimizations.

- Use the Nehalem-EP/Westmere-EP processor specific optimization flag *-xSSE4.2*

  `Warning: Running an executable built with the -xSSE4.2 flag on the Harpertown processors will result in the following error:`

  Fatal Error: This program was not built to run on the processor in your system. The allowed processors are: Intel(R) processors with SSE4.2 and POPCNT instructions support.

If your goal is to have a portable executable that can run on either Harpertown or Nehalem-EP or Westmere-EP, you can choose one of the following approaches:

- use none of the above flags
- use *-xSSE4.1* (with version 11.x and 12.x compilers)
- use *-O3 -ipo -axSSE4.2,xSSE4.1*(with version 11.x and 12.x compilers).

  This allows a single executable that will run on any of the three Pleiades processor types with suitable optimization to be determined at run time.

Turn inlining on: *-ip* or *-ipo*

Use of *-ip* enables additional interprocedural optimizations for single file compilation. One of these optimizations enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

Use of *-ipo* enables multifile interprocedural (IP) optimizations (between files). When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

Use a specific memory model: *-mcmodel=medium* and *-shared-intel*

Should you get a link time error relating to R_X86_64_PC32, add in the compiler option of *-mcmodel=medium* and the link option of *-shared-intel*. This happens if a common block is > 2gb in size.

Turn off all warning messages: *-w -vec-report0 -opt-report0*

Use of *-w* disables all warnings; *-vec-report0* disables printing of vectorizer diagnostic information; and *-opt-report0* disables printing of optimization reports.

Parallelize your code: *-openmp* or *-parallel*

-openmp handles OMP directives and *-parallel* looks for loops to parallelize.

For more compiler/linker options, read **man ifort, man icc**, or

```
%ifort -help
%icc -help
```

With SGI's MPT

DRAFT

This article is being reviewed for completeness and technical accuracy.

Among the many MPI libraries installed on Pleiades, it is recommended that you start with SGI's MPT library.

The available SGI MPT modules are:

```
mpi/mpt.1.25
mpi-sgi/mpt.1.26
mpi-sgi/mpt.2.04.10789
```

There is no default MPT version set, but you are recommended to start with the MPT 2.04.10789 version by loading the *mpi-sgi/mpt.2.04.10789* module. You should load the same module when you build your application on the front-end node and also inside your PBS script for running on the back-end nodes.

Note: Pleiades uses an InfiniBand (IB) network for interprocess RDMA (remote direct memory access) communications and there are two InfiniBand fabrics, designated as *ib0* and *ib1*. In order to maximize performance, SGI advises that the *ib0* fabric be used for all MPI traffic. The *ib1* fabric is reserved for storage related traffic. The default configuration for MPI is to use only the *ib0* fabric.

Environment Variables

When you load an MPT module, several paths (such as CPATH, C_INCLUDE_PATH, LD_LIBRARY_PATH, etc) and MPT or ARRAYD related variables are set or modified. For example, with the mpi-sgi/mpt.2.04.10789 module, the following MPT and ARRAYD related variables are reset to some non-default values:

```
setenv        MPI_BUFS_PER_HOST 256
setenv        MPI_IB_TIMEOUT 20
setenv        MPI_IB_RAILS 2
setenv        MPI_DSM_DISTRIBUTE 0 (for Harpertown processors)
setenv        MPI_DSM_DISTRIBUTE 1 (for Nehalem-EP and Westmere-EP processors)
setenv        ARRAYD_CONNECTTO 15
setenv        ARRAYD_TIMEOUT 180
```

The meanings of these variables and their default values are:

- MPI_BUFS_PER_HOST

  Determines the number of shared message buffers (16 KB each) that MPI is to

allocate for each host (i.e., Pleiades node used in the run). These buffers are used to send and receive long inter-host messages.

Default: 96 pages (1 page = 16KB)

- MPI_IB_TIMEOUT

When an IB card sends a packet it waits some amount of time for an ACK packet to be returned by the receiving IB card. If it does not receive one it sends the packet again. This variable controls that wait period. The time spent is equal to 4 * 2 ^ MPI_IB_TIMEOUT microseconds.

Default: 18

- MPI_IB_RAILS

If the MPI library uses the IB driver as the inter-host interconnect it will by default use a single IB fabric. If this is set to 2, the library will try to make use of multiple available separate IB fabrics (e.g. *ib0* and *ib1*) and split its traffic across them. If the fabrics do not have unique subnet IDs then the rail-config utility is required to have been run by the system administrator to enable the library to correctly use the separate fabrics.

Default: 1

- MPI_DSM_DISTRIBUTE

Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. This feature can also be overridden by using *dplace* or *omplace*. This feature is most useful if running on a dedicated system or running within a cpuset.

Default: Enabled for MPT.1.26; Not Enabled for MPT.1.25

- ARRAYD_CONNECTTO

Tuning this variable is useful when you want to run jobs through arrayd across a large cluster, and there is network congestion. Setting this variable to a higher value might slow down some array commands when a host is unavailable but it will help to prevent MPI start up problems due to connection time-out.

Default: 5 seconds

- ARRAYD_TIMEOUT

Tuning this variable is useful when you want to run jobs through arrayd across a large cluster, and there is network congestion. Setting this variable to a higher value might slow down some array commands when a host is unavailable but it will help to

prevent MPI start up problems due to connection time-out.

Default: 45 seconds

For more MPT related variables, read **man mpi** after loading an MPT module. Some of them may be useful for some applications or for debugging purposes on Pleides. Here are a few of them for you to consider:

- MPI_BUFS_PER_PROC

  Determines the number of private message buffers (16 KB each) that MPI is to allocate for each process (i.e. MPI rank). These buffers are used to send long messages and intrahost messages.

  Default: 32 pages (1 page = 16KB)

- MPI_IB_FAILOVER

  When the MPI library uses IB and a connection error is detected, the library will handle the error and restart the connection a number of times equal to the value of this variable. Once there are no more failover attempts left and a connection error occurs, the application will be aborted.

  Default: 4

- MPI_COREDUMP

  Controls which ranks of an MPI job can dump core on receipt of a core-dumping signal. Valid values are *NONE, FIRST, ALL*, or *INHIBIT. NONE* means that no rank should dump core. *FIRST* means that the first rank on each host to receive a core-dumping signal should dump core. *ALL* means that all ranks should dump core if they receive a core-dumping signal. *INHIBIT* disables MPI signal-handler registration for core- dumping signals.

  Default: FIRST

- MPI_STATS (toggle)

  Enables printing of MPI internal statistics. Each MPI process prints statistics about the amount of data sent with MPI calls during the MPI_Finalize process.

  Default: Not enabled

- MPI_DISPLAY_SETTING

  If set, MPT will display the default and current settings of the environmental variables

controlling it.

Default: Not enabled

- MPI_VERBOSE

  Setting this variable causes MPT to display information such as what interconnect devices are being used and environmental variables have been set by the user to non-default values. Setting this variable is equivalent to passing mpirun the -v option.

  Default: Not enabled

Building Applications

Building MPI applications with SGI's MPT library simply requires linking with -lmpi and/or -lmpi++. See the article SGI MPT for some examples.

Running Applications

```
MPI executables built with SGI's MPT are not allowed to run on the
Pleiades front-end nodes.
```

You can run your MPI job on the back-end nodes in an interactive PBS session or through a PBS batch job. After loading an MPT module, use **mpiexec**, not mpirun, to start your MPI processes. For example:

```
#PBS -lselect=2:ncpus=8:mpiprocs=4:model=har
....
module load mpi-sgi/mpt.2.04.10789
mpiexec -np N ./your_executable
```

The -np flag (with *N* MPI processes) can be omitted if the value of *N* is the same as the product of the value specified for *select* and that specified for *mpiprocs*.

Performance Issues

On Nehalem-EP and Westmere-EP nodes, if your MPI job uses all the processors in each node (i.e, 8 MPI processes/node for Nehalem-EP and 12 MPI processes/node for Westmere-EP), pinning MPI processes greatly helps the performance of the code. SGI's mpi-sgi/mpt.2.04.10789 will pin processes by default by setting the environment variable MPI_DSM_DISTRIBUTE to 1 (or true) when jobs are run on the Nehalem or Westmere nodes. On Harpertown nodes, setting MPI_DSM_DISTRIBUTE to 1 is not recommended due to a processor labeling issue.

If your MPI job do not use all the processors in each node, it is recommended that you disable MPI_DSM_DISTRIBUTE by

```
setenv MPI_DSM_DISTRIBUTE 0
```

and let the Linux kernel decide where to place your MPI processes. If you want to pin processes explicitly, you can use *dplace*. Beware that with SGI's MPT, only 1 shepherd process is created for the entire pool of MPI processes and the proper way of pinning using *dplace* is to skip the shepherd process. In addition, knowledge of the processor labeling in each processor type is essential when you use *dplace*. Below are the recommended ways of pinning an 8 MPI process job with every 4 processes on 4 processor cores of a node:

- Harpertown

```
mpiexec -np 8 dplace -s1 -c2,3,6,7 ./your_executable
```

- Nehalem-EP

```
mpiexec -np 8 dplace -s1 -c2,3,4,5 ./your_executable
```

- Westmere-EP

```
mpiexec -np 8 dplace -s1 -c4,5,6,7 ./your_executable
```

Further information about pinning can be found here.

With MVAPICH

DRAFT

This article is being reviewed for completeness and technical accuracy.

On Pleiades, there are multiple modules of MVAPICH2 built with either gcc or Intel compilers.

```
mpi-mvapich2/1.2p1/gcc
mpi-mvapich2/1.2p1/intel
mpi-mvapich2/1.2p1/intel-PIC
mpi-mvapich2/1.4.1/gcc
mpi-mvapich2/1.4.1/intel
```

The module *mpi-mvapich2/1.2p1/intel-PIC* was built with the -fpic compiler flag.

Building Applications

Here is an example of how to build an MPI application with MVAPICH2:

```
%module load mpi-mvapich2/1.4.1/intel
%module load comp-intel/11.1.072
%mpif90 program.f90
```

Running Applications

To run your job, submit your job through PBS. Within the PBS script, there are two ways to run MPI applications built with MVAPICH2.

1. ```
   #PBS ..
   ...
   module load mpi-mvapich2/1.4.1/intel
   module load comp-intel/11.1.072

   mpiexec -np TOTAL_CPUS your_executable
   ```

2. ```
   #PBS ..
   ...
   module load mpi-mvapich2/1.4.1/intel
   module load comp-intel/11.1.072

   mpirun_rsh -np TOTAL_CPUS -hostfile $PBS_NODEFILE your_executable
   ```

Performance Issues

To pin processes, the MVAPICH library uses the environment variable VIADEV_USE_AFFINITY, which does something similar to SGI's MPI_DSM_DISTRIBUTE.

By default, VIADEV_USE_AFFINITY is set to 1.

If you wish to pin processes explicitly, beware that with MVAPICH, 1 shepherd process is created for each MPI process. You can use the command

```
/u/scicon/tools/bin/qsh.pl jobid \
  'ps -C executable -L -opsr,pid,ppid,lwp,time,comm'
```

to see these processes of your running job. To properly pin MPI processes using *dplace*, one cannot skip the shepherd processes. In addition, knowledge of the processor labeling in each processor type is essential when you use *dplace*. Below are the recommended ways of pinning an 8 MPI process job with every 4 processes on 4 processors of a node:

- Harpertown

  ```
  mpiexec -np 8 dplace -c2,3,6,7 ./your_executable
  ```

- Nehalem-EP

  ```
  mpiexec -np 8 dplace -c2,3,4,5 ./your_executable
  ```

- Westmere-EP

  ```
  mpiexec -np 8 dplace -c4,5,6,7 ./your_executable
  ```

Further information about pinning can be found here.

For more descriptions including the MVAPICH User Guide and other MVAPICH publications, see http://mvapich.cse.ohio-state.edu.

With Intel-MPI

DRAFT

This article is being reviewed for completeness and technical accuracy.

Intel's MPI library is another alternative for building and running your MPI application. The available Intel MPI modules are:

```
mpi-intel/3.1.038
mpi-intel/3.1b
mpi-intel/3.2.011
```

To use Intel MPI, first create a file $HOME/.mpd.conf that has the single line:

```
MPD_SECRETWORD=sometext
```

('sometext' should be unique for each user)

and change the permission of the file to read/write by you only.

```
%chmod 600 $HOME/.mpd.conf
```

Building Applications

To compile, load an Intel compiler module and an Intel MPI module. Make sure that no other MPI module is loaded (i.e., MPT, MVAPICH or MVAPICH2)

```
%module load mpi-intel/3.1.038
%module load comp-intel/11.1.072
```

Use the **mpiifort**/**mpiicc** scripts which invoke the Intel ifort/icc compilers.

```
%mpiifort -o your_executable program.f
```

Running Applications

To run it, in your PBS script make sure the intel MPI modules are loaded as above, start the MPD daemon, use mpiexec, and terminate the daemon at the end. For example,

```
#PBS ..
..
module load mpi-intel/3.1.038
module load comp/intel/10.1.021_64

# Note: The following three lines should really be in one line
```

```
mpdboot --file=$PBS_NODEFILE --ncpus=1 --totalnum=`cat $PBS_NODEFILE  |
sort -u | wc -l` --ifhn=`head -1 $PBS_NODEFILE`
 --rsh=ssh --mpd=`which mpd` --ordered

# CPUS_PER_NODE and TOTAL_CPUS below represent numerical numbers
# for the job at hand

mpiexec -ppn CPUS_PER_NODE -np TOTAL_CPUS ./your_executable

# terminate the MPD daemon

mpdallexit
```

With OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

Building Applications

To build an OpenMP application, you need to use the *-openmp* Intel compiler flag:

```
%module load comp-intel/11.1.072
%ifort -o your_executable -openmp program.f
```

Running Applications

The maximum number of OpenMP threads an application can use on a Pleiades node depends on (i) the number of physical processor cores in the node and (ii) if hyperthreading is available and enabled. Hyperthreading technology is not available for the Harpertown processor type. It is available and enabled at NAS for the Nehalem-EP and Westmere-EP processor types. With hyperthreading, the OS views each physical core as two logical processors and can assign two threads to it. This is beneficial only when one thread does not keep the functional units in the core busy all the time and can share the resources in the core with another thread. Running in this mode may take less than 2 times the walltime compared to running only 1 thread on the core.

```
Before running with hyperthreading for your production runs, it is
recommended that you experiment with it to see if it is beneficial
for your application.
```

	Maximum Threads	
Processor Type	Maximum Threads without Hyperthreading	Maximum Threads with Hyperthreading
Harpertown	8	N/A
Nehalem-EP	8	16
Westmere-EP	12	24

Here is sample PBS script for running OpenMP applications on a Pleiades Nehalem-EP node without hyperthreading:

```
#PBS -lselect=1:ncpus=8:ompthreads=8:model=neh,walltime=1:00:00

module load comp-intel/11.1.072

cd $PBS_O_WORKDIR
```

```
./your_executable
```

Here is sample PBS script with hyperthreading:

```
#PBS -lselect=1:ncpus=8:ompthreads=16:model=neh,walltime=1:00:00

module load comp-intel/11.1.072

cd $PBS_O_WORKDIR

./your_executable
```

With SGI's MPI and Intel OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

Building Applications

To build an MPI/OpenMP hybrid executable using SGI's MPT and Intel's OpenMP libraries, your code needs to be compiled with the *-openmp* flag and linked with the *-mpi* flag.

```
%module load comp-intel/11.1.072 mpi-sgi/mpt.2.04.10789
%ifort -o your_executable prog.f -openmp -lmpi
```

Running Applications

Here is a sample PBS script for running MPI/OpenMP application on Pleiades using 3 nodes and on each node, 4 MPI processes with 2 OpenMP threads per MPI process.

```
#PBS -lselect=3:ncpus=8:mpiprocs=4:model=neh
#PBS -lwalltime=1:00:00

module load comp-intel/11.1.072 mpi-sgi/mpt.2.04.10789
setenv OMP_NUM_THREADS 2

cd $PBS_O_WORKDIR

mpiexec ./your_executable
```

You can specify the number of threads, *ompthreads*, on the PBS resource request line, which will cause the PBS prologue to set the OMP_NUM_THREADS environment variable.

```
#PBS -lselect=3:ncpus=8:mpiprocs=4:ompthreads=2:model=neh
#PBS -lwalltime=1:00:00

module load comp-intel/11.1.072 mpi-sgi/mpt.2.04.10789

cd $PBS_O_WORKDIR

mpiexec ./your_executable
```

Performance Issues

For pure MPI codes built with SGI's MPT library, performance on Nehalem-EP and Westmere-EP nodes improves by pinning the processes through setting MPI_DSM_DISTRIBUTE envrionment variables to 1 (or true). However, for MPI/OpenMP codes, all the OpenMP threads for the same MPI process have the same process ID and setting this variable to 1 causes all OpenMP threads to be pinned on the same core and the

performance suffers.

It is recommended that MPI_DSM_DISTRIBUTE is set to 0 and *omplace* is to be used for pinning instead.

If you use Intel version 10.1.015 or later, you should also set KMP_AFFINITY to *disabled* or OMPLACE_AFFINITY_COMPAT to *ON* as Intel's thread affinity interface would interfere with dplace and omplace.

```
#PBS -lselect=3:ncpus=8:mpiprocs=4:ompthreads=2:model=neh
#PBS -lwalltime=1:00:00

module load comp-intel/11.1.072 mpi-sgi/mpt.2.04.10789

setenv MPI_DSM_DISTRIBUTE 0
setnev KMP_AFFINITY disabled

cd $PBS_O_WORKDIR

mpiexec -np 4 omplace ./your_executable
```

With MVAPICH and Intel OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

Building Applications

To build an MPI/OpenMP hybrid executable using MVAPICH and Intel's OpenMP libraries, use *mpif90, mpicc, mpicxx* with the *-openmp* flag.

```
%module load comp-intel/11.1.072  mpi-mvapich2/1.4.1/intel
%mpif90 -o your_executable prog.f90 -openmp
```

Running Applications

With MVAPICH, a user's environment variables (such as VIADEV_USE_AFFINITY and OMP_NUM_THREADS) are not passed in to mpiexec, thus they need to be passed in explicitly, such as with /usr/bin/env.

Here is an example on how to run a MVAPICH/OpenMP hybrid code with a total of 12 MPI processes and 2 OpenMP threads per MPI process:

```
#PBS -lselect=3:ncpus=8:mpiprocs=4:model=neh

module load comp-intel/11.1.072 mpi-mvapich2/1.4.1/intel

mpiexec /usr/bin/env VIADEV_USE_AFFINITY=0 OMP_NUM_THREADS=2 ./your_executable
```

Performance Issues

Setting the environment variable VIADEV_USE_AFFINITY to 0 disables CPU affinity because MVAPICH does its own pinning. Setting it to 1 actually causes multiple OpenMP threads to be placed on a single processor.

Porting to Columbia

Default or Recommended compiler version and options

DRAFT

This article is being reviewed for completeness and technical accuracy.

Intel compiler versions 10.0, 10.1, 11.0 and 11.1 are available on Columbia as modules. Use the 'module avail' command to find available versions.

The current default compiler module on Columbia is *intel-comp.10.1.013*.

In addition to the few flags mentioned in the article Recommended Intel Compiler Debugging Options, here are a few more to keep in mind:

Turn on optimization: *-O3*

If you do not specify an optimization level (*-On*, *n=0,1,2,3*), the default is *-O2*. If you want more aggressive optimizations, you can use *-O3*. Note that using *-O3* may not improve performance for some programs.

Turn inlining on: *-ip* or *-ipo*

Use of *-ip* enables additional interprocedural optimizations for single file compilation. One of these optimizations enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

Use of *-ipo* enables multifile interprocedural (IP) optimizations (between files). When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

Parallelize your code: *-openmp* or *-parallel*

-openmp handles OMP directives and *-parallel* looks for loops to parallelize.

For more compiler/linker options, read **man ifort, man icc**, or

```
%ifort -help
%icc -help
```

Porting to Columbia: With SGI's MPT

DRAFT

This article is being reviewed for completeness and technical accuracy.

The available SGI MPT modules on Columbia are:

```
mpt.1.16.0.0
mpt.1.18.0.0
mpt.1.19.0.0
mpt.1.22.0.0
mpt.1.25
```

The current default version is *mpt.1.16.0.0*.

Environment Variables

On Columbia, when you load any of the above MPT modules, several environment variables such as CPATH, INCLUDE, LD_LIBRARY_PATH, etc., are modified by pre-pending the appropriate MPT directories. Also, the following MPT-related environment variables are modified from their default values for improved performance:

```
setenv MPI_BUFS_PER_HOST 256
setenv MPI_BUFS_PER_PROC 256
setenv MPI_DSM_DISTRIBUTE
```

The meanings of these variables and their default values are:

- MPI_BUFS_PER_HOST

  Determines the number of shared message buffers (16 KB each) that MPI is to allocate for each host (i.e., C21, C22, C23, C24). These buffers are used to send and receive long inter-host messages.

  Default: 32 pages (1 page = 16KB) for mpt.1.16, mpt.1.18, mpt.1.19, mpt.1.22
  Default: 96 pages (a page = 16KB) for mpt.1.25

- MPI_BUFS_PER_PROC

  Determines the number of private message buffers (16 KB each) that MPI is to allocate for each process. These buffers are used to send long messages and intra-host messages.

  Default: 32 pages (1 page = 16KB)

- MPI_DSM_DISTRIBUTE (toggle)

  Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the host with which that CPU is associated. This feature can also be overridden by using dplace or omplace. This feature is most useful if running on a dedicated system or running within a cpuset.

  Default: Not enabled

Building Applications

Building MPI applications with SGI's MPT library simply requires linking with -lmpi and/or -lmpi++. See the article SGI MPT for some examples.

Running Applications

```
MPI executables built with SGI's MPT are not allowed to run on the
Columbia front-end node.
```

You can run your MPI job on C21 - C24 in an interactive PBS session or through a PBS batch job. Use **mpiexec** (under /PBS/bin) or mpirun to start your MPI processes. For example:

```
#PBS -lncpus=8
....
mpiexec -np N ./your_executable
```

The -np flag (with *N* MPI processes) can be omitted if the value of *N* is the same as the value specified for *ncpus*.

Porting to Columbia: With OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

Building Applications

To build an OpenMP application, you need to use the -openmp Intel compiler flag:

```
%ifort -o your_executable -openmp program.f
```

Note that if you are compiling separate files, then -openmp is required at the link step to link in the OpenMP library.

Running Applications

Note that *OMP_NUM_THREADS* is set to 1 by default for PBS jobs. Reset it to the number of threads that you want.

Here is a sample PBS script for running OpenMP applications on Columbia:

```
#PBS -lncpus=8,walltime=1:00:00

setenv OMP_NUM_THREADS 8

cd $PBS_O_WORKDIR

./your_executable
```

Porting to Columbia: With MPI and OpenMP

DRAFT

This article is being reviewed for completeness and technical accuracy.

Building Applications

To build a hybrid MPI+OpenMP application, you need to compile your code with the -openmp compiler flag and link in both the Intel OpenMP and the SGI MPT library:

```
%ifort -o your_executable -openmp program.f -lmpi
```

Running Applications

Process/thread placement is critical to the performance of MPI+OpenMP hybrid codes. Two environment variables should be set to get the proper placement:

- MPI_DSM_DISTRIBUTE

  Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. Currently, the CPUs are chosen by simply starting at relative CPU 0 and incrementing until all MPI processes have been forked.

- MPI_OPENMP_INTEROP

  Setting this variable modifies the placement of MPI processes to better accommodate the OpenMP threads associated with each process. For this variable to take effect, you must also set MPI_DSM_DISTRIBUTE.

Also note that *OMP_NUM_THREADS* is set to 1 by default for PBS jobs. Reset it to the number of threads that you want.

Here is a sample PBS script for running MPI+OpenMP hybrid (2 MPI processes, 4 OpenMP threads per MPI process) applications on Columbia:

```
#PBS -lncpus=8,walltime=1:00:00

setenv MPI_DSM_DISTRIBUTE
setenv MPI_OPENMP_INTEROP
setenv OMP_NUM_THREADS 4

cd $PBS_O_WORKDIR

mpirun -np 2 ./your_executable
```